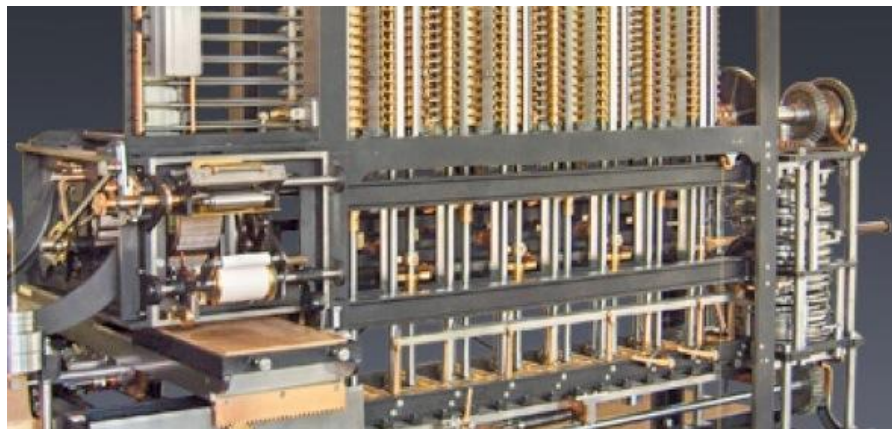


- Surcharge des opérateurs
- Héritage



Cours 2

Alice Cohen-Hadria
(Transparents B. Gas & ACH)

alice.cohenhadria@gmail.com

Surcharge des opérateurs

- Surcharge des opérateurs
- Héritage

On repart de l'exemple avec la Classe complexe.

```
class Complex {
    double x, y;
public :
    void Complex( double a=0, double b=0) {
        x = a;
        y = b;
    }
    Complex add(const Complex c) {
        return Complex (x + c.get_x(), y + c.get_y());
    }
};

void main() {
    Complex c1(1,1), c2(2,2), c3;

    c3 = c1.add(c2);
}
```

Surcharge des opérateurs

- Surcharge des opérateurs
- Héritage

On voudrait écrire :

```
void main() {  
    Complex c1(1,1), c2(2,2), c3;  
  
    c3 = c1 + c2;  
}
```

On va surcharger l'opérateur +

Rappel surcharge :

Deux fonctions qui ont le même nom mais pas la même signature.

On va donc garder les opérations de type $2 + 3$, ou $1.0 + 3.0$, et ajouter le comportement $c1 + c2$ à l'opérateur +.

Quel opérateur est appliqué dépend du type des opérandes.

Notation préfixe et infixe

▸ Surcharge des
opérateurs

▸ Héritage

Notation préfixe :

- $2 + 3$
- L'opérateur $+$ est appelé avec les paramètres 2 et 3

Notation infixe :

- $+ 2 3$
- On indique d'abord l'opérateur est ensuite ses paramètres. Comme avec une fonction $f(x, y)$ ou $+(2, 3)$

Pour la surcharge des opérateurs

Pour désigner l'opérateur $+$:

- On écrit : `operator+` en notation infixe, et on va coder son comportement.
- Ensuite on peut utiliser l'opérateur $+$ en notation préfixe :
 - Complexe $c1, c2, c3$;
 - $c3 = c1 + c2$;

Scope (portée) et ::

- Surcharge des opérateurs
- Héritage

On utilise l'opérateur :: pour préciser dans quel scope (une portée) une fonction ou une variable se trouve.

```
class Complex {
    double x, y;
public :
    Complex( double a=0, double b=0) {
        x = a;
        y = b;
    }
    void f(); // déclaration seule
};
```

```
void Complex::f() { . . . };
```

Pour la surcharge des opérateurs

On peut définir cette surcharge dans la portée de la classe complexe.

Surcharge des opérateurs

- Surcharge des opérateurs
- Héritage

```
class Complex {
    double x, y;
public :
    void Complex( double a=0, double b=0) {
        x = a;
        y = b;
    }
    Complex operator+(const Complex&) const; // surcharge de l'opérateur +
};

Complex Complex::operator+(const Complex &c) const {
    return Complex(x + c.x, y + c.y);
}

void main() {
    Complex c1(1,1), c2(2,2), c3;

    c3 = c1 + c2;    // EQV à c3 = c1.operator+(c2);
}
```

Surcharge des opérateurs

Surcharge à l'extérieur de la classe: lorsque l'on n'a pas accès à la classe:

```
class Complex {
    double x, y;
public :
    double X() const {return x;}           // devient nécessaire
    double Y() const {return y;}           // devient nécessaire
    Complex( double a=0, double b=0) {
        x = a;
        y = b;
    }
};

Complex operator+(const Complex &c1, const Complex &c2) {
    return Complex(c1.X() + c2.X(), c1.Y() + c2.Y());
}

void main() {
    Complex c1(1,1), c2(2,2), c3;

    c3 = c1 + c2;      // EQV à c3 = operator+(c1,c2);
}
```

Surcharge des opérateurs - Cas des flux

- Surcharge des opérateurs
- Héritage

```
class Complex {
    double x, y;
public :
    void Complex( double a=0, double b=0) {
        x = a;
        y = b;
    }
    void afficher () {
        cout << '(' << x << ',' << y << ')';
    }
};
```

```
void main() {
    Complex c1(1,1), c2(2,2), c3;
    cout << 'Nombre c1';
    c1.afficher();
    cout << endl;
}
```

→ En une seule ligne avec la surcharge.

Surcharge des opérateurs - Cas des flux

- Surcharge des opérateurs
- Héritage

On veut écrire :

```
void main() {
    Complex c1(1,1);
    cout << "c1=" << c1 << endl;
    // EQV à cout << "c1=" << operator<<(cout, c1) << endl;
}
```

On va surcharger l'opérateur <<. Cet opérateur est dans la classe ostream. On ne veut pas modifier cette classe. On va définir la surcharge **à l'extérieur** de la classe :

```
ostream& operator<<(ostream &o, const Complex &c) {
    o << '(' << c.get_x() << ',' << c.get_y() << ')';
    return o;
}
```

Le paramètre o est passé par référence. Le type de retour est ostream&.
On est à l'extérieur de la classe, on a besoin des accesseurs pour accéder aux attributs.

Surcharge des opérateurs - Cas des flux

- Surcharge des opérateurs
- Héritage

Les attributs de Complex sont private (par défaut). On doit rajouter les accesseurs à la class Complex:

```
class Complex {
    double x, y;
public :
    double get_x() const {return x;}
    double get_y() const {return y;}
    void Complex( double a=0, double b=0) {
        x = a;
        y = b;
    }
};
```

get_x et get_y doivent être const. On a passé un const Complex& à la surcharge de <<. Si les accesseurs ne sont pas const:

passing 'const Complex' as 'this' argument discards qualifiers
[-fpermissive] |

Surcharge des opérateurs - Opérateurs booléens

- Surcharge des opérateurs
- Héritage

```
class Complex{
    private :
        double x, y;
    public :
        Complex(double a=0, double b=0) { x = a; y = b;}
        int get_x() const { return x;}
        int get_y() const { return y;}
        bool operator==(const Complex&) const;
};

bool Complex::operator==(const Complex &c) const {
    return (x == c.x) && (y == c.y);
}

int main(){
    Complex c1(1,1), c2(2,2);
    bool b = c1 == c2;
    cout << "c1 == c2" << b << endl;
    return 0;
}
```

Surcharge des opérateurs - Opérateurs booléens

- Surcharge des opérateurs
- Héritage

On peut aussi évidemment surcharger les autres opérateurs booléens : $<$, $>$, $<=$, $>=$.

Dans le cas des nombres complexes, ces opérateurs n'ont pas de sens.

Surcharge des opérateurs - opérateurs somme et affectation

- Surcharge des opérateurs
- Héritage

```
class Complex{
    private :
        double x, y;
    public :
        Complex( double a=0, double b=0) {
            x = a;
            y = b;
        }
        int get_x() const { return (*this).x;}
        int get_y() const { return (*this).y;}
        void set_x(double a) { x=a; }
        void set_y(double b) { y=b; }
        void operator+=(const Complex);
};
```

```
void Complex::operator+=(const Complex c) {
    set_x(x+c.x);
    set_y(y+c.y);
}
```

```
int main()
{
    Complex c1(1,1), c2(2,2);
    c1 += c2;
    cout << "c1 = " << c1 << endl;

    return 0;
}
```

Surcharge des opérateurs - Avertissements

- Surcharge des opérateurs
- Héritage

Les performances d'un programme peuvent être grandement affectées par des surcharges d'opérateurs mal écrits.

En particulier, il faut faire attention à la copie d'objets (qui peut être coûteuse).
Quand cela est possible, utiliser des passages par référence !

L'Héritage: définition

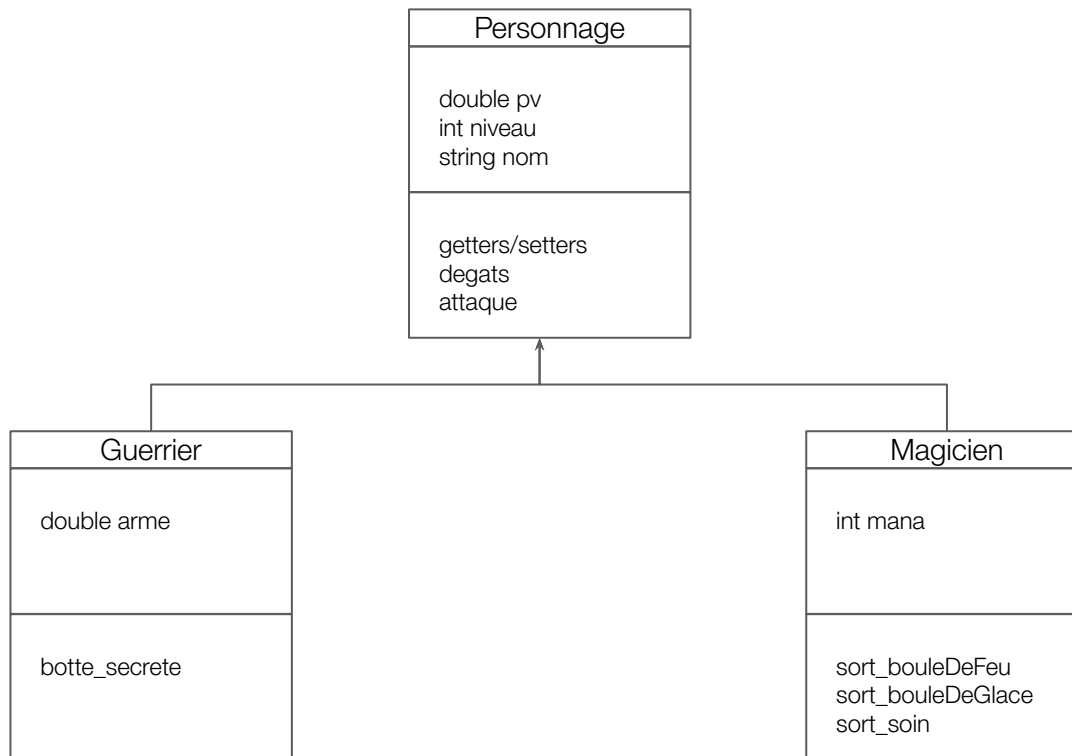
▸ Surcharge des
opérateurs

▸ Héritage

- Définition d'une nouvelle classe par réunion d'une ou plusieurs classes préexistantes appelées **classes de base** ou **classes mères**.
- La nouvelle classe est appelée la **classe fille**.
- Correspond à la relation «est une sorte de ...». Exemple
 - Un Chat est une sorte d'Animal (la classe chat hérite de la classe Animal)
 - Un Chirurgien est une sorte de Docteur
- L'héritage peut être **simple** ou **multiple** → Une classe peut hériter d'une ou plusieurs classes.

L'Héritage: Exemple

- Surcharge des opérateurs
- Héritage



L'Héritage: Accessibilité

- Surcharge des opérateurs
- Héritage

- Trois modes de dérivation: **private**, **protected** et **public**
- Permettent de déterminer l'accessibilité des membres
- Les objets des classes de base sont toujours présents dans les classes dérivées

L'héritage par défaut est l'héritage **private**

```
class class_derivee : private classe_de_base { ... }
```

```
class class_derivee : protected classe_de_base { ... }
```

```
class class_derivee : public classe_de_base { ... }
```

L'Héritage - Déclaration

- Surcharge des opérateurs
- Héritage

```
class Personnage{  
    private:  
        double pv;  
        int niveau;  
        string nom;  
};  
  
class Guerrier : public Personnage  
  
class Magicien : public Personnage
```

L'Héritage: visibilité des membres

- Surcharge des opérateurs
- Héritage

La visibilité des membres des classes de base et dérivées :

- Les noms des membres des classes dérivées masquent les noms des membres de la classe de base.
- On peut néanmoins accéder aux membres de la classe de base, avec la notation de portée ::

```
int main() {  
    Magicien m(100, 10, "Alice", 40);  
    cout << m.get_pv() << endl;  
    cout << m.Personnage::get_pv() << endl;  
    return 0;  
}
```

L'Héritage public

- Surcharge des opérateurs
- Héritage



CLASSE DE BASE	CLASSE DERIVEE
public	public
protected	protected
private	inaccessible
inaccessible	

- L'interface de la classe de base reste accessible aux concepteurs des classes dérivées, ainsi qu'aux utilisateurs des classes dérivées

L'Héritage public

- Surcharge des opérateurs
- Héritage

```
class Base {
    private:
        int pvt = 1;

    protected:
        int prot = 2;

    public:
        int pub = 3;


        // function to access private member
        int getPVT() {
            return pvt;
        }
};

class PublicDerived : public Base {
    public:
        // function to access protected member from
        Base
        int getProt() {
            return prot;
        }
};
```

```
int main() {
    PublicDerived object1;
    cout << "Private = " <<
    object1.getPVT() << endl;
    cout << "Protected = " <<
    object1.getProt() << endl;
    cout << "Public = " << object1.pub <<
    endl;
    return 0;
}
```

L'Héritage protégé

- Surcharge des opérateurs
- Héritage



CLASSE DE BASE	CLASSE DERIVEE
public	protected
protected	
private	inaccessible
inaccessible	

- L'implémentation de la classe de base reste accessible aux concepteurs des classes dérivées, mais pas aux utilisateurs des classes dérivées

L'Héritage protégé

- Surcharge des opérateurs
- Héritage

```
class Base {
private:
    int pvt = 1;

protected:
    int prot = 2;

public:
    int pub = 3;

    // function to access private member
    int getPVT() {
        return pvt;
    }
};


class ProtectedDerived : protected Base {
public:
    // function to access protected member from Base
    int getProt() {
        return prot;
    }

    // function to access public member from Base
    int getPub() {
        return pub;
    }
};
```

```
int main() {
    ProtectedDerived object1;
    cout << "Private cannot be accessed."
    << endl;
    cout << "Protected = " <<
    object1.getProt() << endl;
    cout << "Public = " <<
    object1.getPub() << endl;
    return 0;
}
```

L'Héritage privé

- Surcharge des opérateurs
- Héritage



CLASSE DE BASE	CLASSE DERIVEE
public	private
protected	
private	inaccessible
inaccessible	

- Si D dérive de B, alors un objet D est une sorte de B mais les utilisateurs de D n'ont pas à le savoir

L'Héritage privé

- Surcharge des opérateurs
- Héritage

```
class Base {
    private:
        int pvt = 1;

    protected:
        int prot = 2;

    public:
        int pub = 3;

        int getPVT() {
            return pvt;
        }
};

class PrivateDerived : private Base {
    public:
        int getProt() {
            return prot;
        }

        int getPub() {
            return pub;
        }
};
```

```
int main() {
    PrivateDerived object1;
    cout << "Private cannot be accessed." <<
endl;
    cout << "Protected = " <<
object1.getProt() << endl;
    cout << "Public = " << object1.getPub() <<
endl;
    return 0;
}
```

L'Héritage privé

- Surcharge des opérateurs
- Héritage

- L'interface de la classe de base disparaît au profit de l'interface de la classe dérivée
- Il devient obligatoire de réécrire l'interface de la classe de base dans la classe dérivée

La classe mère ne fournit pas le comportement attendu pour les classes filles.

Si on a :

```
class Magicien : private Personnage
```

Alors :

```
int main(){
    Magicien m(100, 10, "Alice", 40);
    cout << m.get_pv() << endl;
    cout << m.Personnage::get_pv() << endl;
    return 0;
}
```

error: 'int Personnage::get_pv() const' is inaccessible within this context

L'Héritage : constructeurs et destructeurs

- Surcharge des opérateurs
- Héritage

- Lors de la construction d'un objet les constructeurs des classes de base sont implicitement ou explicitement appelés.

- Syntaxe :

```
class(parametres)
: classe_de_base1(parametres1), classe_de_base2(parametres2), ... {
    ...
}
```

- Les destructeurs des classes de base sont toujours appelés (rappel: ils sont uniques)
- Les constructeurs des classes de base sont d'abord appelés
- Les constructeurs des objets sont ensuite appelés
- Les instructions du corps du constructeur sont ensuite exécutées

L'Héritage : Exemple

- Surcharge des opérateurs
- Héritage

```
Personnage(double pv=100, int niveau=0, string nom=""){
    this->pv = pv;
    this->niveau = niveau;
    this->nom = nom;
}
```

Pour les classes filles :

```
Guerrier(double pv=100, int niveau=0, string nom="", double arme=10) :
    Personnage(pv, niveau, nom){
    this-> arme = arme;
}
```

```
Magicien(double pv=100, int niveau=0, string nom="", int mana=1) :
    Personnage(pv, niveau, nom){
    this-> mana = mana;
}
```

L'Héritage : appel des destructeurs

- Surcharge des opérateurs
- Héritage

- Lors de la destruction d'un objet les instructions du destructeur sont d'abord exécutées
- Ensuite sont appelés les destructeurs des objets membres
- Ensuite sont appelés les destructeurs des classes de base dans l'ordre inverse de leurs déclarations
- Enfin l'espace mémoire est libéré.

L'Héritage : Exemple

- Surcharge des opérateurs
- Héritage

```
class Personnage{
private:
    double pv;
    int niveau;
    string nom;

public:
    Personnage(double pv=100, int
niveau=0, string nom=""){
        this->pv = pv;
        this->niveau = niveau;
        this->nom = nom;
    }

    string get_nom() const {return nom;}
    int get_pv() const {return pv;}
    int get_niveau() const {return
niveau;}

    void set_pv(double p){pv = p;}

    void degats(double d){
        pv -= d;
    }
    void attaque(Personnage&);
};
```

```
class Guerrier : public Personnage {
private:
    double arme;

public:
    Guerrier(double pv=100, int niveau=0, string
nom="", double arme=10) : Personnage(pv, niveau, nom){
        this-> arme = arme;
    }

    void attaque (Personnage &cible){
        cible.degats(arme);
    }
};
```

L'Héritage : Exemple

- Surcharge des opérateurs
- Héritage

```
class Magicien : public Personnage{
private:
    int mana;
public:
    Magicien(double pv=100, int niveau=0, string nom="", int mana=1) : Personnage(pv, niveau, nom){
        this->mana = mana;
    }

    double sort_bouleDeFeu() const{
        return 100.;
    }

    double sort_bouleDeGlace() const{
        return 150.;
    }

    void sort_soin(Personnage &p) {
        p.set_pv(p.get_pv() + 10);
    }

    void attaque(Personnage &p){
        double proba = rand() / (float) RAND_MAX;
        if (proba < 0.5){
            p.degats(sort_bouleDeFeu());
        }else{
            p.degats(sort_bouleDeGlace());
        }
    }
};
```

L'Héritage : redéfinition des fonctions membres

- Surcharge des opérateurs
- Héritage

- Lorsqu'une fonction membre d'une classe dérivée a le même nom qu'une fonction membre de la classe de base, cette dernière est masquée à l'utilisateur de la classe dérivée.

```
int main() {

    Personnage p(100, 10, "Alice");
    Guerrier g(100, 10, "Alice2", 40);
    Magicien m(100, 10, "Alice3", 40);

    m.attaque(g);

    return 0;
}
```

Méthode de la classe magicien appelée (et pas celle de la classe Personnage).

L'Héritage : redéfinition des fonctions membres

- Surcharge des opérateurs
- Héritage

- Lorsqu'une fonction membre d'une classe dérivée a le même nom qu'une fonction membre de la classe de base, cette dernière est masquée à l'utilisateur de la classe dérivée.

```
int main() {

    Personnage p(100, 10, "Alice");
    Guerrier g(100, 10, "Alice2", 40);
    Magicien m(100, 10, "Alice3", 40);

    m.attaque(g);

    return 0;
}
```

Signature de la méthode attaque de la classe Magicien :

```
void attaque(Personnage &p)
```

Comme Guerrier hérite de Personnage : pas de problème.

L'Héritage : héritage multiple

- Surcharge des opérateurs
- Héritage

Une classe peut hériter de plusieurs classes mères.

Syntaxe:

```
class C : public A, public B
```

Gestion des membres homonymes (attributs et méthodes) des classes A et B à l'intérieur de la classe C:

Opérateur de gestion de portée ::

Si A et B ont toutes les deux un attributs att, s'en référer avec A::att et B::att